Electronic Notes in Theoretical Computer Science 39 No. 3 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume39.html 20 pages

Verification of the legOS Scheduler using Uppaal

Lone Halkjaer, Karen Haervi, Anna Ingolfsdottir [1]

Institute for Computer Science, Aalborg University, Aalborg, Denmark
{halkjaer,kh,annai}@cs.auc.dk

Abstract

This article concentrates on the scheduler in the operating system legOS. LegOS is an open source embedded operating system for the Lego Mindstorms® system. The scheduler in legOS practices starvation of lower priority threads. In this article the validity of starvation problems is proven through tests of the scheduler and through an Uppaal model of the scheduler wherein the starvation is verified. A new scheduler is designed and modeled in Uppaal. This Uppaal model is used to verify that starvation is no longer a problem in the new design. The new design is implemented in a new scheduler and tests are performed to show that the problem with starvation is no longer present.

1 Introduction

In this article the scheduler of the legOS operating system will be in focus. The legOS operating system which is part of the Mindstorms® system operates on a RCX™ [2] that has an embedded Hitachi H8 processor. The legOS operating system has features normal to an operating system such as semaphores, dynamic memory management and scheduling etc.

Operating systems have schedulers to decide which process the cpu is to service next. Schedulers are designed with different emphasis on which areas are important. The area of importance can be that every process gets the same fair service or that the processes needing short time at the cpu get serviced before processes needing longer time at the cpu [4].

The scheduler implemented in the legOS operating system schedules the processes according to their priority. These priorities are defined externally to the operating system, which means that it is the programmer, that defines the priority a process is going to have.

[1] Special thanks to Luca Aceto for help and support
[2] A programmable micro controller

The scheduler in legOS has starvation of lower priority threads, which means that when the programmer assigns a low priority to a process to be run on the legOS operating system, this process will not be allowed any processor time, after being initiated if there is a process of higher priority willing to run. This is an unfortunate situation since the task the starved process was supposed to perform will never be performed [6].

This can be seen as a severe problem since the legOS operating system is not properly documented and a programmer would not necessarily know about the problem with starvation in legOS.

We came across the problem of starvation when we were making some test programs to run on the RCX™ and could not explain the behavior of the tasks being performed by the programs. We therefore started to look into the code of the operating system and found the problem to be the priority scheduler.

The problem with starvation can be solved by designing a new scheduler and implementing it in the legOS operating system. The new scheduler is designed and from this design we have developed a Uppaal model and this model is used as a base for the implementation of the new scheduler. We have tested the new scheduler to show that starvation does not occur.

Outline of the article: Section 2 presents the tools used in connection to the work with the legOS scheduler. Section 3 outlines the problem with the legOS scheduler and describes the Uppaal model and the tests performed to clarify the starvation problem. Section 4 describes the design of a new scheduler implemented in legOS, the new Uppaal model and presents the tests that have been performed with the new scheduler. Section 5 summarizes the results given in this article.

2 Preliminaries

This section contains a brief overview of the tools used.

2.1 The Lego Mindstorms®

The Lego Mindstorms® [3] is a system developed by Lego® and the Massachusetts Institute of Technology. It provides an easy way to program robots, build using standard Lego bricks, some sensors, motors and a programmable micro controller, called the RCX™. The heart of the RCX™ is an embedded Hitachi H8 processor.

Programs can be downloaded to the RCX™ via an infrared link and a PC and are typically constructed using a graphical [4] language, the RCX™ code.

[3] For further information see http://www.legomindstorms.com/

[4] Graphical language, meaning that you literally build a program using animated Lego bricks with different functionality

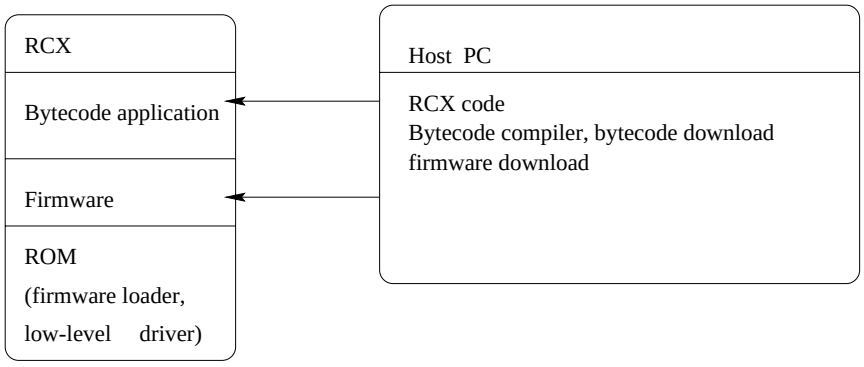

Fig. 1. text

It is also possible to program using a C-like programming language called Not Quite C (NQC). For related work where NQC and Uppaal has been used see [5]. Figure 1 illustrates the RCX™ software system, which consists of a graphical programming environment and a bytecode virtual machine.

Though the system provided with the Mindstorms set (the Robotics Invention System) is easy to use, it is limited in many ways. The graphical environment limits variable usage to one counter and there is no task synchronization, no stack, no real heap and no concept of priorities [1] This is a reason for looking at a more powerful operating system for the Mindstorm® robots. One such is legOS.

2.2 The legOS operating system

LegOS is an open source alternative to the firmware that is part of the Mindstorm system. It was developed by Markus L. Noga whose goal was to develop a small, embedded operating system implemented in C/C++ and assembler.

LegOS offers a number of features including preemptive multitasking, energy saving, dynamic memory management, semaphores, native access to display, buttons, infrared communication, motors, sensors and a new kernel. There is no file system since there is no mass storage on the RCX, but the operating system has the three basic functionalities of an operating system: Memory management, Resource management and Task management. The last function will be the issue of this article.

LegOS programs can be developed using standard C/C++ or assembler and when designing legOS Markus L. Noga chose to comply with the standard UNIX interfaces wherever possible, thus making it easy to get started with

legOS for a large group of programmers already familiar with C/C++ and UNIX [5] [1].

The version of legOS considered in this article is 0.2.3, the newest version at the time of writing. The reason for choosing this version is that legOS, with this version, finally has become a "real" operating system. Earlier the programs developed for legOS were compiled together with the operating system and the whole thing was downloaded to the RCX. In the 0.2.3 version the operating system is downloaded once and programs developed are downloaded individually afterwards.

The legOS documentation consist of only a few outdated articles and the API [6], outdated meaning that the documentation is for older releases of legOS.

The legOS compiler can easily be set up under Linux, using the GNU gcc compiler, but we were unsuccesfull in installing the 0.2.3 version under both Solaris and Windows [7]

2.3 Uppaal

Uppaal [8] was first released in 1995 and is developed by Uppsala University in Sweden and Aalborg University in Denmark.

Uppaal is an integrated tool containing an environment for modeling and verifying real-time simulated systems modeled as timed automata, extended with data types. The main application areas of Uppaal are in real-time controllers and real-time systems where the purpose often is to model and analyze existing systems [2]. But Uppaal is appropriate for all systems that can be modeled as a collection of non-deterministic processes with finite control structures and real-valued clocks, communicating through channels and shared variables [3].

Uppaal provides both a graphical user interface(GUI) and textual format for the description language. In the GUI it is possible to describe the system in networks of timed automata. Uppaal performs typechecks on the timed automata and the user can thereafter choose to use the verifier or the simulator on the timed automata, see figure 2.

The verifier in Uppaal is able to handle both liveness properties and reachability properties. Reachability can be checked using $\forall\Box$ and $\exists\Diamond$. For $\forall\Box\beta$ to be satisfied all reachable states must satisfy β. For $\exists\Diamond\beta$ to be satisfied some reachable state must satisfy β. When using the verifier, Uppaal delivers a boolean answer and shows a diagnostic trace to support the answer whenever possible. The tool also assists in simple debugging where it reports all inconsistent states and all deadlocked states.

[5] LegOS contains a reduced Unix library with e.g. execi
[6] Application Programming Interface listing functions and complicated datastructures
[7] For further information see http://www.noga.de/legOS/
[8] For further information see http://www.uppaal.com

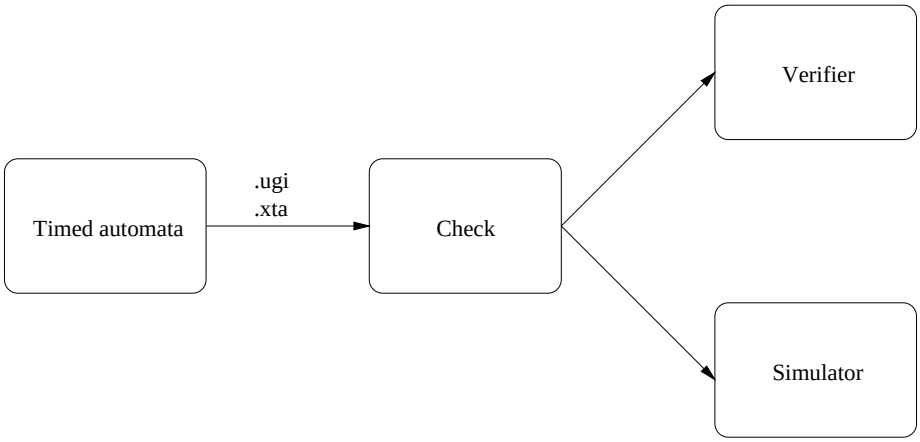

Fig. 2. The Uppaal tools

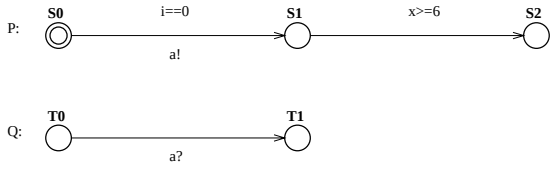

Fig. 3. A simple Uppaal model

2.4 Modeling in Uppaal

Figure 3 demonstrates how models are made in Uppaal. It is a simple Uppaal model containing the timed automata P and Q. The automaton P has three states (S0, S1, S2) and the integer variables `i, x` and an action channel `a`. The automaton Q has two states (T0, T1) and is synchronised with P on action channel `a`. P takes the transition S0 to S1 when the integer variable `i` is equal to 0 and at the same time it synchronizes with Q through the action channel `a`. The transition between S1 and S2 is taken when `x`¿=6.

3 Scheduler

In this section the details of the original legOS scheduler are presented. First the details of how the actual legOS scheduler is implemented is described. Next comes a subsection detailing the tests which show the problem with the way the scheduler works and after that a subsection describes how the scheduler is modeled in Uppaal. Finally the Uppaal model is presented and used to verify that starvation is a problem in the scheduler.

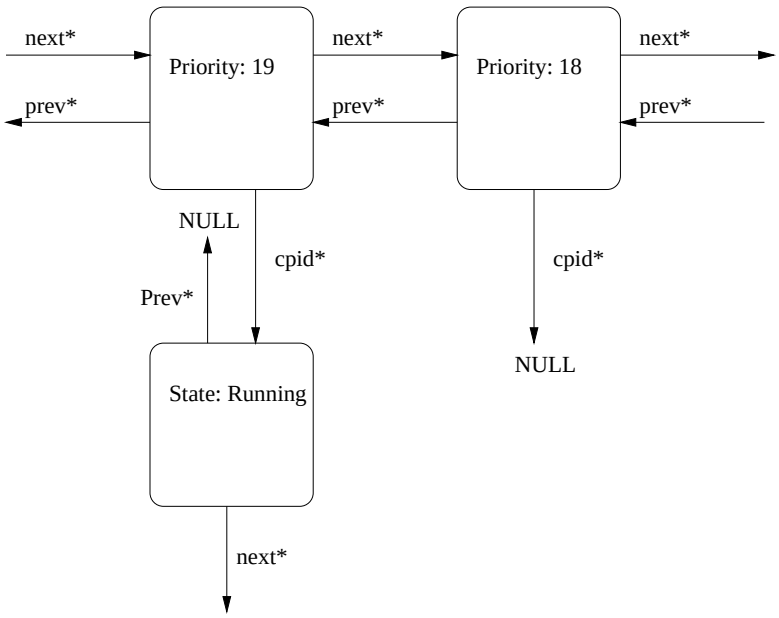

Fig. 4. The legOS priority list

3.1 The legOS Scheduler

The scheduler in legOS works with priority scheduling. The processes can be assigned a priority from 1 to 20, where 20 is the highest.

The scheduler is implemented as a list of lists. More specifically there is a priority list in which each node holds a pointer to a list where every process has the same priority, see figure 4. Investigating the scheduling algorithm of legOS reveals a problem with starvation of processes, because the scheduler always starts at the head of the priority list, when finding the next process to run. The code for this can be seen below.

```
// find next process willing to run
priority = priority_head;
```

The pointer is assigned to the pointer to the head of the priority list. If some high priority process is always willing to run, processes with lower priority are never given any time at the processor.

3.2 Test of the legOS Scheduler

To illustrate and test the scheduling algorithm in legOS we have developed a test program and used it in six tests where the priority of the processes were varied. The purpose of the tests are to confirm the starvation of lower priority threads in legOS.

The test setup consists of a program running four threads, `display_thread`, `stop_thread`, `motorA_thread` and `motorB_thread`. Each thread does a different thing in order to observe which threads are actually running.

The `display_thread` outputs to the display in turn two things "Hello"

Test 1:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 5. Test 1 of the original scheduler

and "test". The `stop_thread` kills the processes whenever the run button is pressed thereby returning "power" to the operating system. The two motor threads turn the two motors on and off (one motor each) in turn, every 10 sec.

Since the legOS operating system is preemptive[9] it can not be expected to behave exactly the same way every time. This is because it is not possible to say exactly when a process is going to be suspended. For this reason every test is performed 10 times and the result are given in figures 5 through 10.

Having performed the tests we can see that eventhough the operating system is preemptive the test program performed alike every time it was run. This is probably because of the simplicity of the test program.

The test figures, figure 5 to 10 show in the first column the number of the test and the names of the threads. In the second column the priorities are given, in the third column, `initial run`, it is indicated whether or not the thread is initiated. In the last column it is shown if the thread is running according to its programming or if it is running inconsistently with its programming e.g. when a motor thread does not turn off every 10 sec.

The results are given so that a `1` signifies a `yes` the thread is running according to its assignment. A `0` signifies a `no`, this is when the thread is not getting any processor time, meaning that the thread is not running.

In figure 5 the `motorA_thread` is given a low priority whereas the three other threads are all given priority 20. It can be seen from the test result that all 4 threads get initialised but that `motorA_thread` is not getting any processor time after that. This can be seen in the test program by motor A being started when starting the program but then the motor does not perform the 10 sec. shutdown like it is supposed to, like motor B. In figure 6 the test is repeated with the `display_thread` as the one being starved. The difference here is that the `display_thread` is here given priority 10 whereas the other three threads are given priority 20. This test has exactly the same behavior,

[9] The strategy of allowing processes that are logically runnable to be temporarily suspended is called "preemptive scheduling" and is in contrast to the "run to completion" strategy, also called "non-preemptive scheduling"[7]

Test 2:	Priority	Initial run	Runing normal
Display_thread	10	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 6. Test 2 of the original scheduler

Test 3:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 7. Test 3 of the original scheduler

Test 4:	Priority	Initial run	Runing normal
Display_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 8. Test 4 of the original scheduler

starving the low priority thread the `display_thread`.

From test 3 and 4, see figure 7 and figure 8, it can be seen that when assigning all threads the same priority the threads are assigned equal time at the processor, meaning that all threads are running according to their programming. These tests were performed with one test where all threads had priority 20 and one test where all threads had priority 2.

Test 5:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0

Fig. 9. Test 5 of the original scheduler

Test 6:	Priority	Initial run	Runing normal
Display_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	0-0-0-0-0 0-0-0-0-0

Fig. 10. Test 6 of the original scheduler

Test 5 and 6, see figure 9 and figure 10, are included to show that the operating system will starve any number of threads (two threads in test 5 and three threads in test 6) with lower priority. This can be seen by all threads getting initialised in both tests but only the highest priority processes being allowed to continue their normal run.

These tests show that lower priority threads do not get any processor time whenever a high priority thread is willing to run.

3.3 Making the Uppaal model of the Scheduler

This subsection describes how the Uppaal model of the legOS scheduler is made. The model is made by examining the code and conducting tests to confirm how the scheduler operates.

Whenever making a model from some code you have to abstract away from some of the details. This makes the model more usable and easier to understand. Without the abstractions the model would not be any less complex than the code and the purpose of the model would be gone. The abstractions can be a source of error, because you might accidently abstract away something important. This fact has to be taken into considerations when building a

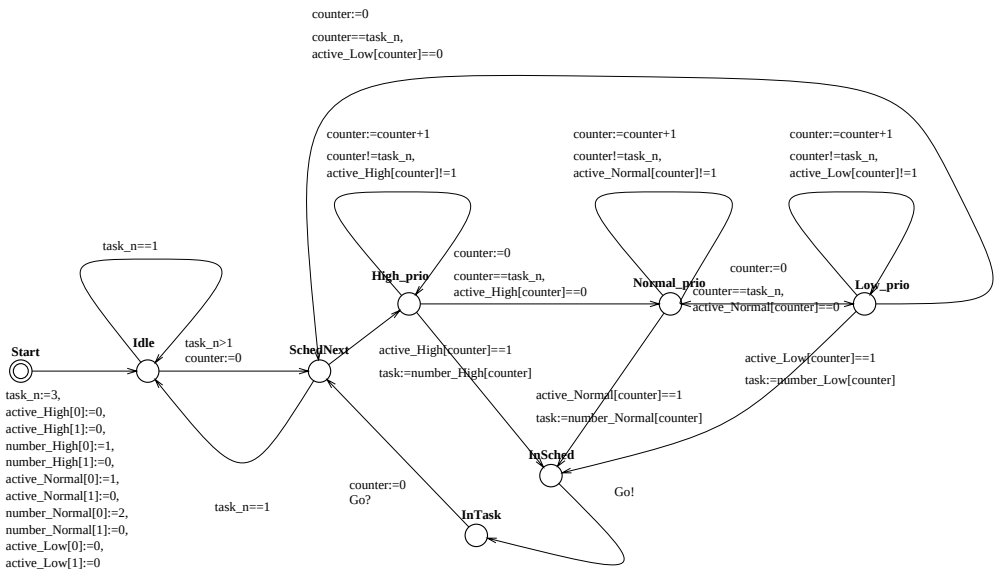

Fig. 11. The Uppaal model of the scheduler

model. For this reason the abstractions made when making the Uppaal model of the legOS scheduler are described here.

In the model presented here we have abstracted away some of the states of the processes. In the model the processes can only be in one of two states, either they would like to run or not. In the implementation the processes can be in 5 different states: `running` (is on the cpu), `waiting` (waiting for some event), `sleeping` (wants to get on the cpu), `zombie` (the scheduler needs to remove it from the scheduling queue) or `dead` (this state is never in use [10]).

Some of the priorities are abstracted away. In the model only three priorities `High`, `Normal` and `Low` are considered. In the implementation you can assign priorities 1 through 20. The Uppaal model of the scheduler can be seen in figure 11

3.4 The Uppaal model of the Scheduler

The scheduled processes are stored in arrays because Uppaal does not support lists. To ease the reading of the model, only three priority queues are modeled, i.e. the states `High_prio`, `Normal_prio` and `Low_prio`.

The priority list is modeled as two types of arrays, one that holds the information on whether the process wants to run or not and one containing the tasknumber of the scheduled tasks. The first type of array, `active_High`, `active_Normal` and `active_Low`, can only contain 1 or 0, indicating whether the process wants to run (1) or not (0).

[10] We can not explain this feature from the code that we have investigated

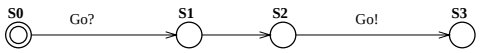

Fig. 12. An instruction

The second type of array, `number_High`, `number_Normal`, `number_Low` holds the task numbers of the scheduled tasks. Here the labels `High`, `Normal` and `Low` indicates the priority of the processes in the array. The counter "`counter`" keeps track of which process in the array is currently investigated. The cell number of a process in the array is the same, meaning that evaluating the second cell in both arrays will evaluate the same process, its task number and whether it wants to run or not. A counter `task_n` holds the number of all tasks scheduled.

Looking more specifically at what happens when simulating the model. First the `Idle` process is initiated (`task_n` := 1). If no other processes are scheduled, the operating system will run this process (The idle process has the lowest priority). When `task_n` becomes larger than 1 the scheduler moves to SchedNext to see which process to run next.

First it evaluates the array, holding processes with the highest priority. If the counter becomes equal to the maximum number of tasks (`task_n`) without finding a process willing to run, it proceeds to the state `Normal_prio` to look for a process to run, by searching through the normal priority arrays. If one is not found it moves to the state `Low_prio` and starts its search in that array. If no process is found it returns to the `SchedNext` state, now ready to either go into the `Idle` state if `task_n` == 1 or start looking for a process to run.

When a process willing to run is found, the scheduler moves to the `Insched` state, `task` is assigned to the process number found and the instruction with that process number is given a `Go` signal. A model of an instruction can be seen in figure 12. The process is then allowed to run for a timeslice (default is 20 msec), where after it signals `Go` to the scheduler and the scheduler goes into the `SchedNext` state. The scheduler then evaluates if more tasks are scheduled if not it goes into the `Idle` state else it goes to the `High_prio` state, to find the next process to run.

3.5 Verification of Starvation

The Uppaal verification tool, verifyta, has been used here to verify the problem with starvation in the legOS scheduler. The verification is done by the use of logic inquiries to Uppaal. Uppaal will give a boolean answer and will show a diagnostic trace to support the answer whenever possible.

The traces chosen to run in Uppaal are described in this section. The premises are a setup of the scheduler where three processes are scheduled, one

is scheduled with the highest priority. All three processes are willing to run.

The first inquiry put to Uppaal is: Will the scheduler ever enter state `Normal_prio` or state `Low_prio`.

- ∃◇ `scheduler.Normal_prio or scheduler.Low_prio`

  `Property is not satisfied`

The answer that the property is not satisfied indicates that the scheduler will not enter the states `Normal_prio` or `Low_prio` whenever a process of higher priority is willing to run like it is the case in this verification setup.

The next inquiry is: Can processes of low priority (priority normal or low) ever get the `Go` signal from the scheduler (being scheduled at the processor).

- ∃◇ `(ins2.S1) or (ins3.S1)`

  `Property is not satisfied`

This indicates that the scheduler will not send the `Go` signal to any process of lower priority when there is a process of high priority willing to run.

Finally: Will any of the processes in the system ever get scheduled at the processor (get in the S1 state).

- ∃◇ `(ins1.S1)or(ins2.S1)or(ins3.S1)`

  `Property is satisfied`

That this inquiry is satisfied indicates that the scheduler will schedule at least one of the processes at the processor.

From these logical inquiries Uppaal shows, that the scheduler will never get into the states of lower priorities and therefore never let processes of lower priority run. Uppaal also shows that a process can run. Investigating the trace Uppaal offers, shows that this process will always be the one with the highest priority.

4 The new Scheduler

The new scheduler designed for the legOS operating system is designed with inspiration from [4]. The design and implementation of the new scheduler is performed by first outlining a design of the scheduler see subsection 4.1. We have made a Uppaal model of the scheduler, verified that the new design works e.g. that it does not have starvation, and then implemented the new scheduler in C. This newly implemented scheduler is then integrated in the legOS operating system.

4.1 Design of the new Scheduler

The scheduler will be implemented with priority queues, where 20 is the highest priority and 1 is the lowest priority like in the scheduler already implemented in legOS. The difference will be that the operating system will increase

the priority of the oldest process waiting for processor time just before finding the next process to run. This will prevent complete starvation from ever happening, in that every process eventually will get the highest priority.

Every process will be assigned an externally defined priority that will decide which priority queue the process will be placed in when initialised.

The legOS operating system will timestamp every process when it is initialised. Every time the scheduler searches through the priority queue for a new process to schedule it will look for the oldest process and increase its priority by one. When a up-prioritised process has been scheduled at the processor its priority will be returned to the original setting and the process starts its climb up the priorities once more.

4.2 Making the new Uppaal model

When making this model, like making the Uppaal model of the original scheduler, we have had to make some simplifications and abstractions compared to how the scheduler is going to be implemented in C. These abstractions are made because of problems like moving a process in an array to a different array when upgrading the priority.

Since Uppaal does not support lists, processes are stored in arrays. This means that a major difference between the Uppaal model and the way the scheduler is going to be implemented are in the searching through the arrays. In a list you would normally insert a member at the back of the list and retrieve it from wherever it is located. In an array you have to search through the array to find an empty location, then insert the process in the place, when retrieving it you do the same as with lists.

The Uppaal model of the new scheduler is made with the Uppaal model of the original model as a base. Therefore only the changes and additions in the new model are included in the next subsection. A model of the new scheduler can be seen in figure 13.

4.3 The Uppaal model of the new Scheduler

First the changes to the scheduler automaton are described. The clock `Gcl` is a global clock which is used to timestamp the processes. It is implemented as an integer variable because Uppaal does not support assignment of clocks to integers thereby eliminating the possibility of timestamping. The new states `resched` and `reschedule` participates in the rescheduling of the oldest process. The syncronisation channels `Reschedule`, `Stamp` and `Upgrade` are added. `Upgrade` starts the rescheduling of the oldest process and `Reschedule` starts the rescheduling of a process. The `instruction` automaton from the model of the original scheduler is not changed in the new model. But a new automaton, `processes` is introduced in order to manage the more complicated states of a process, see figure 14.

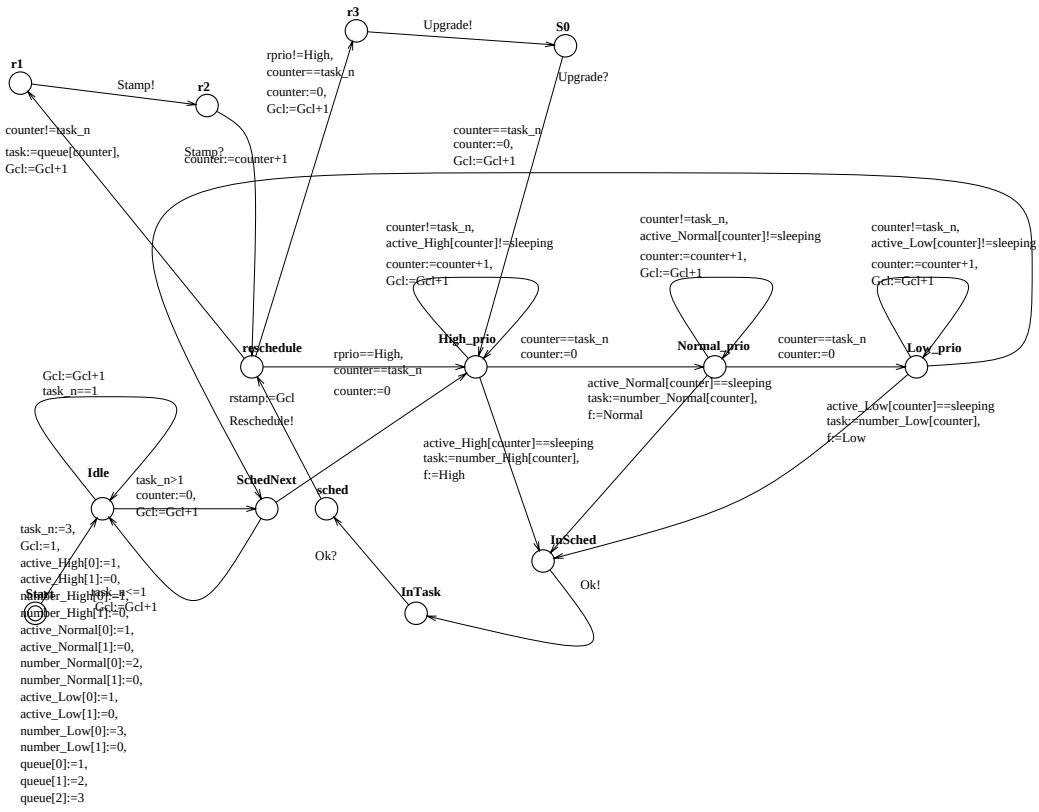

Fig. 13. State diagram of the new scheduler

The automaton, `processes` will be described next. The variable `privstamp` is a timestamp internal to the process which is given when the process is initialised. This timestamp is assigned the value of `Gcl` every time the process has had time at the processor. The `privprio` is the private priority of a process and the variable `state` indicates which state a process is in. A process can be in one of several states; `sleeping` (wants to run), `running` (in the processor) or `waiting` (waiting for some event to happen). The state `instruction` sends the signal to the `instruction` automaton. The states `upgrade` and `reschedule` participates in the upgrading and rescheduling of the process.

When simulating a run on the Uppaal model and more than one process are scheduled, the scheduler model moves to the `ShedNext` state and directly from there to the `High_prio` state.

When finding a process willing to run the scheduler model will move to the `InShed` state and signal `Ok` to the process with the tasknumber found.

When scheduled, the process is sent a `Ok` signal and it will go from the `Sleeping` state to the `Running` state and run some instructions until the timeslice is up. This is done by sending `go` signals to the instruction. When the timeslice is used, the process will get a new timestamp, go into the `Sleeping`

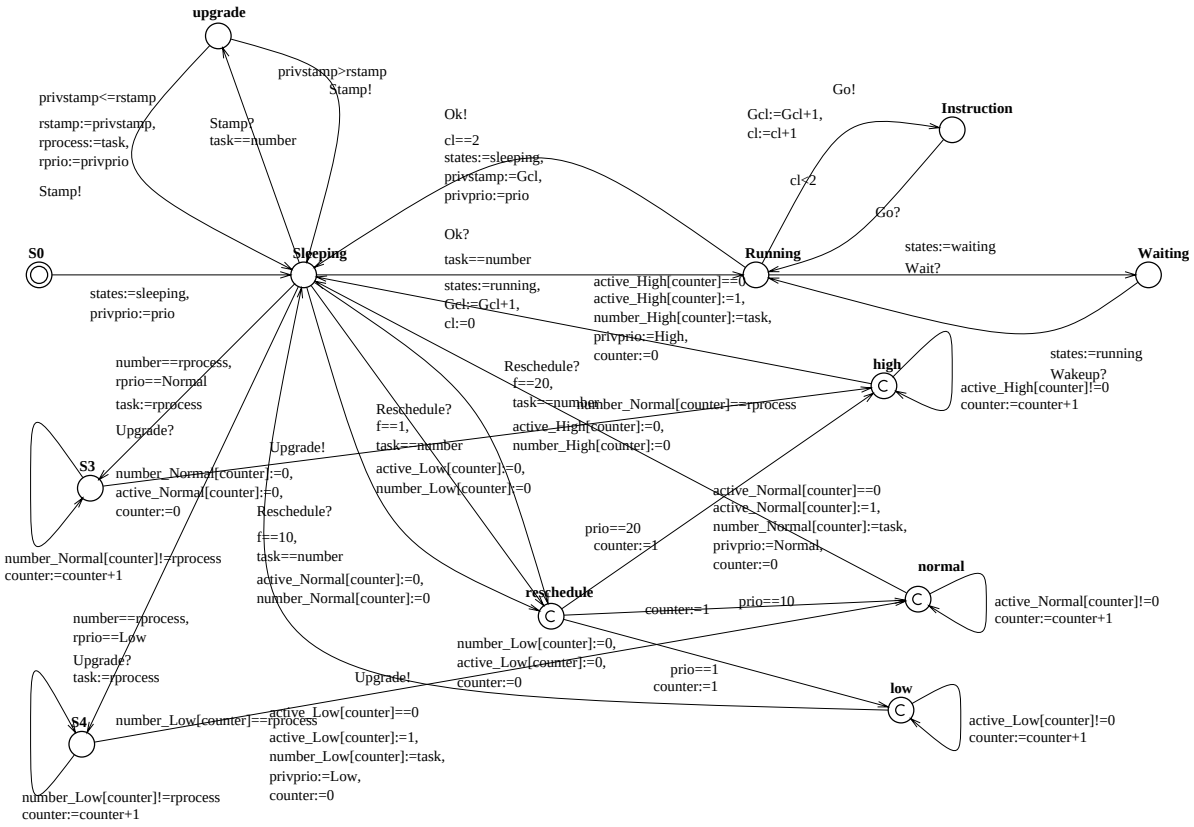

Fig. 14. A model of a process

state and then signal `Ok` and return to the `sched` state of the scheduler.

Before the scheduler finds the next process to run it starts its rescheduling mechanism. First it signals `Reschedule` to allow the process that has just run to erase the array cell it was just in and to put itself back in the array it was originally in. The original value of the priority of a process is stored in the `privprio` variable. Next the scheduler sends the `Stamp` synqronisation and the automaton `processes` will run through every process in all the queues waiting in the scheduler, comparing timestamp until it has found the process with the oldest timestamp. The scheduler will then return the `Stamp` synqronisation and later send the signal `Upgrade` to the oldest process and the process is then upgraded i.e. it is moved to the priority array one priority above the array it was in earlier.

4.4 Verification of the Uppaal model of the new Scheduler

We have performed verification on the Uppaal model of the new scheduler in order to prove that the scheduler could not possibly perform starvation on any of the waiting processes.

The inquiries put to Uppaal are logically the same as where put to Uppaal in connection with the verification of the original scheduler, see subsection 3.5. The setup is also the same.

The premises of the verification are: the new scheduler is set up to run with three processes. One of these processes has high priority and the others have lower priority. All the three processes are willing to run.

In order for Uppaal to answer the first inquiry we have had to put an upper bound on the variabel `Gcl` containing the global clock. This is done to keep the global state-space from being infinite.

The first inquiry asks if the scheduler will ever enter the `Normal_prio` state or the `Low_prio` state. The scheduler would normally enter this state whenever it is searching through the array of processes with normal or low priority.

- $\exists\Diamond$ `scheduler.Normal_prio or scheduler.Low_prio`

  `Property is not satisfied`

  The answer that the property is not satisfied indicates that the scheduler will still not enter the `Normal_prio` or the `Low_prio` state if a high priority process is willing to run, but this is no longer a problem since the processes are upgraded as time passes.

The second inquiry is whether or not the processes not having the highest priority when initiated, will ever enter the running state.

- $\exists\Diamond$ `(p2.Running) or (p3.Running)`

  `Property is satisfied`

  This property is satisfied because the scheduler will eventually send the `Ok` signal to a process initiated with a normal or low priority. This result is different from the result seen in the original scheduler and can be explained by the scheduler upgrading the priority of the lower priority processes and thereby eventually putting them in the front of the priority queue.

The last inquiry is asking the scheduler if it will ever send the `Ok` signal to any of the three processes.

- $\exists\Diamond$ `(p1.Running) or (p2.Running) or (p3.Running)`

  `Property is satisfied`

  The scheduler indicates that it will not starve all the processes, so eventually it will give a `Ok` signal to one of the processes willing to run. The diagnostic trace shows that the process that is given a `Ok` signal is always a process with the highest priority.

The second inquiry shows that the scheduler will send the `Ok` signal to processes not initiated with the highest priority. This verifies that the Uppaal model of the new scheduler is not performing starvation.

4.5 Implementation of the new Scheduler

The new scheduler in legOS is also implemented as a list of lists. The first list is a priority list where each node is assigned a priority and holds a pointer to a list with that priority. In addition to the original scheduler, each node holds a private priority and a time stamp. When a process is started the private priority is assigned the value of the priority given to the process and the timestamp is assigned the value of the system time.

Two things happen after the scheduler is done scheduling a process, before it starts looking for a new process to run. The process that has just run is given a new timestamp and put back in the priority queue where it was originally placed. The private priority variable holds the value of the original priority. Furthermore the scheduler locates the oldest process by comparing timestamps and upgrades the oldest by putting it in the priority list with one higher priority.

There are three processes that are never upgraded namely the `main`, the `idle` and the `scheduler` process. The way the new scheduler finds the next process to run works just as in the original scheduler, where the scheduler searches through the priority lists from the highest priority to the lowest until finding a process willing to run.

4.6 Test of the new Scheduler

After having implemented the new scheduler in C according to the overall design plan and the Uppaal model made, we have performed some test on the scheduler to see if it works as expected. The testing that we have performed are identical to the tests performed on the original scheduler which are documented in subsection 3.2.

In the first two tests, see figure 15 and 16, one thread in each test is assigned a lower priority than the other three threads. The results show that the new scheduler does not starve the lower priority threads, but instead every thread is running. We explain this through the way the scheduler is implemented where although low priority threads have to climb up the priorities before being allowed time at the processor this fact can not be seen by an observer since we are talking about differences of only a few microseconds.

Test 3 and 4, see figures 17 and 18 have been performed because of the wish to have identical tests on both the original scheduler and the new scheduler. In both of these tests all threads are assigned the same priority. In test 3 all threads have priority 20 and in test 4 all threads have priority 2. The results show that all threads are running normal, which means that the scheduler is scheduling the processor time equally.

From test 5, see figure 19 and test 6, see figure 20 it can again be seen that all threads are getting time at the processor. The test performed in test 5 are with two threads having a lower priority, in test 6 three threads having

Test 1:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 15. Test 1 of the new scheduler

Test 2:	Priority	Initial run	Runing normal
Display_thread	10	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 16. Test 2 of the new scheduler

Test 3:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 17. Test 3 of the new scheduler

a lower priority.

The conclusion on the tests performed on the new scheduler must be that the scheduler works as planned. The lower priority threads are getting less processor time but the difference between the time the low processes and the time the high processes are given are negligible and can not be observed.

Test 4:	Priority	Initial run	Runing normal
Display_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 18. Test 4 of the new scheduler

Test 5:	Priority	Initial run	Runing normal
Display_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 19. Test 5 of the new scheduler

Test 6:	Priority	Initial run	Runing normal
Display_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
Stop_thread	20	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorA_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1
MotorB_thread	2	1-1-1-1-1 1-1-1-1-1	1-1-1-1-1 1-1-1-1-1

Fig. 20. Test 6 of the new scheduler

5 Conclusion

In this paper the scheduler in the legOS operating system has been examined through tests and with a simulation and verification of a Uppaal model of the scheduler. We found that the scheduler was performing unsatisfactory i.e. it performs starvation of lower priority processes as soon as a process of higher

priority was willing to run.

This problem has been the center of this article where a new scheduler has been designed, modeled in Uppaal, implemented and tested. Uppaal has been used as a useful tool for helping to verify that the new scheduler is not performing starvation.

From the test we have performed on the new scheduler we must conclude that the behavior of the new design is working without starvation of lower priority threads when running programs on the RCX-brick. The scheduler does not perform starvation but instead it gives less processor time to processes of low priority, this must be considered the reason of giving some processes a lower priority than others.

References

[1] Noga, Markus L., "Designing the legOS Multitasking Operating System," http://www.noga.de/legOS/, 1999.

[2] Bengtson, Johan, and Larsen, Kim G., and Larsson, Fredrik, andPetterson, Paul, and Yi, Wang, and Weise, Carsten, "New Generation of Uppaal," http://www.docs.uu.se/docs/rtmv/uppaal/ documentation.html, 1995.

[3] Larsen, Kim G., and Petterson, Paul, and Yi, Wang, "UPPAAL in a Nutshell," http://www.docs.uu.se/docs/rtmv/uppaal/ documentation.html, 1995.

[4] Silberschatz, Abraham, and Galvin, Peter Baer Galvin, "Operating System Concepts," ISBN: 0471364142, 1998.

[5] Iversen, Torsten K., and Kristoffersen, Kre J., and Larsen, Kim G., and Laursen, Morten, and Madsen, Rune G., and Mortensen, Steffen K., and Petterson, Paul, and Thomasen, Chris B., "Model-Checking Real-Time Control Programs," http://www.docs.uu.se/docs/rtmv/uppaal/documentation.html, 1999.

[6] Stallings, William, "Operating Systems" ISBN: 0131809776, 1995.

[7] Tanenbaum, Andrew S., "Modern Operating Systems," ISBN: 0135881870, 1992.